Abstract — The task of middleware is to mask out problems of heterogeneity and distribution for application developers. With the emergence of new application domains, like multimedia and real-time applications, flexible support of QoS and real-time requirements becomes a major challenge. Reconfigurable bindings and policy driven binding protocols are the major means in the MULTE framework to support these requirements. Our experience with flexible protocol configuration has shown that the above requirements cannot be solved only by dynamic (re-)configuration of communication protocols. Additionally, flexibility in establishing and managing bindings is necessary. We have implemented a toolkit that allows configurable signalling, including tasks such as QoS negotiation, resource reservation, connection management, monitoring, etc. The toolkit is used in the MULTE-ORB to support various functional and non-functional application requirements.

Keywords — Configurable Signalling, Flexible Protocols, Middleware, Policies, QoS

I. Introduction

Middleware has emerged as a central architectural component in supporting distributed applications and services. The role of middleware is to present a higher level programming paradigm for application writers (typically object-oriented or, more recently, component-based) and to mask out problems of heterogeneity and distribution. New application domains for middleware technology, like real-time systems, embedded systems, fault-tolerant systems, multimedia, and mobility, introduce additional problems to be solved by the middleware. In this context, flexible and extensible Quality-of-Service (QoS) and real-time support is one of the major challenges. Flexibility and extensibility are important, because a single fixed middleware solution will not be able to support requirements of the new application domains, like

• Dynamic QoS support: applications should be able to specify QoS requirements and to change them dynamically. The middleware should provide the requested QoS and adapt to changes in QoS requirements, resource availability, etc.
• Policy control: the middleware should enable end-users, application developers, and system managers to specify policies for QoS mapping, negotiation, monitoring, adaptation, etc.
• Support for seamless system evolution: the integration of new components to make use of new system services (e.g., new signalling protocols in the Internet) should not require changes of existing components and middleware entities.

In order to support these requirements, our binding framework is prototyped in the MULTE-ORB by integrating the flexible protocol framework Da CaPu in the CORBA 2.0 implementation COOL. Our experience with this prototype has shown that the above requirements cannot be solved only by dynamic (re-)configuration of communication protocols. Additionally, flexibility in establishing and managing bindings is necessary. The trade-off between guaranteed QoS and minimal latency is a typical example for this need: distributed real-time applications require that all necessary resources are allocated before the application peers exchange data. The execution of the resource reservation protocol obviously increases the time it takes to establish the corresponding binding. In contrast, interactive distributed multimedia applications are interested to exchange data as fast as possible. Thus, the immediate exchange of application data and the parallel offline execution of signalling protocols like QoS negotiation and resource reservation might be a good solution for several application scenarios.

The basic problem that prohibits such a type of flexibility in middleware systems is the insufficient separation of mechanisms and policies, which is in most other systems, like operating systems, a major design guideline. In our context of signalling, the mechanisms correspond to the basic support for exchange of signalling information, and the policies define the signalling protocols. It should be noted that we use the term signalling in this work with a broad a sense and include QoS management tasks, like QoS mapping, admission control, QoS negotiation, adaptation, etc., and binding management tasks, like binding establishment, binding release, and binding reconfiguration.

With the general goal of separation between signalling mechanisms and policies in mind, we have developed a toolkit for the MULTE-ORB that allows dynamic configuration of signalling protocols. The toolkit represents the basic signalling mechanism and enables the MULTE-ORB to execute user defined signalling policies. These policies comprise two components:

• an extensible set of signalling modules, each implementing a single signalling or management protocol, and
• a script that specifies which signalling modules in which order must be executed by the toolkit.

As a result, the MULTE-ORB is able to support dynamically changing QoS requirement according to user defined policies. It is even possible to support differ-
ent QoS requirements for binding establishment, data exchange, and binding release. Furthermore, new signalling protocols can be easily integrated, tested, and utilized, without changing other components of the MULTE-ORB. In this paper, we focus on design, implementation, and evaluation of the toolkit. The remainder of this paper is structured as follows, Section II introduces the MULTE binding framework and Section III describes the original signalling solution in the MULTE-ORB 1.0 and its drawbacks. In Section IV, we explain the architecture of the management toolkit and Section V describes its implementation and preliminary evaluation. A brief discussion of related work and an overview of our ongoing and our future work is given in the concluding Section VI.

II. THE MULTE BINDING FRAMEWORK

In this section, we briefly introduce the MULTE binding framework. The binding framework defines all entities involved in the creation and manipulation of bindings, as well as the binding components and the way they are handled. Some principles are assumed for the binding framework, such as the support for multi-party and multimedia bindings, QoS negotiation/re-negotiation among the binding endpoints, and the use of policies. Policies are an important part of the framework since they provide a flexible way to control the behaviour of the entities involved in binding actions. Furthermore, it should be noted that the binding framework is based upon the concept of open binding.

A. Basic Elements

The binding framework defines the following entities:

- Binding factory (BF): responsible for creation of bindings. The BF is a distributed entity, meaning the existence of replicated factories. Each of these replicas behaves according to well-defined roles when co-operating to create a binding.
- Binding manager (BM): responsible for monitoring and co-ordinating binding adaptation such as adding, removing or replacing components in a binding, as well as changing the behaviour of existing components, e.g., by changing resource reservations for the components. Furthermore, the BM can ask the application to adapt.
- Binding destructor (BD): releases bindings and frees the allocated resources.
- Binding template (BT): specifies the properties, constraints, and configuration guidelines for binding objects. Binding templates are used by BFs to define the structure of new bindings.
- Configuration: the set of components that constitute the internals of binding objects. A configuration is the product of the evaluation of a BT, and is represented by an object graph depicting the identities and types of the binding components, and their interconnections.
- Binding protocol (BP): protocols used by BFs to create new bindings, by BM to adapt existing bindings, and by BDs to remove bindings.

BFs, BMs, and BDs are collectively referred to as lifecycle components of bindings. In contrast, binding components are the basic building blocks of binding configurations, and are defined as objects that can have multiple distinct interfaces.

B. Binding Protocols

Different types of binding protocols are used by BFs, BMs, and BDs during the life cycle of bindings. All binding protocols have some common features, due to the way open bindings are structured. First, binding protocols guide the communication and co-ordination among the BF, BM, and BD replicas involved in a binding. Second, binding protocols combine a set of simple protocols for tasks like QoS management, connection management, resource management, and other signalling tasks. For all these tasks mostly multiple alternatives with different properties exist. For example, QoS negotiation can be performed as unilateral, bilateral, or triangular negotiation; and the binding can be established in a three-way handshake, two-way handshake, or implicit. These alternatives are combined according to user policies. Thus, the behaviour of a binding protocol is determined by policies and specified in scripts.

By defining distinct binding protocols, we define different ways in which bindings may be created, and adapted. This allows the creation of bindings with different characteristics (e.g., multi-party, with QoS guarantees). In addition, different binding protocols may be optimised for different binding templates. Therefore, prior to requesting the creation (and subsequent modification) of a binding, the user may set up policies that configure the life cycle components of the binding such that they behave according to the desired binding protocol.

III. SIGNALLING IN MULTE-ORB 1.0

Our first prototype implementation, called MULTE-ORB 1.0, is based on the flexible protocol framework Da CaPo (Dynamic Configuration of Protocols), and the CORBA 2.0 implementation COOL. Da CaPo serves as transport protocol and is used instead of TCP/IP. Da CaPo enables us to configure implementations of protocol functions of arbitrary granularity to complex protocols. Naturally, the implementation of a protocol function corresponds to a component in the binding framework.

Furthermore, we have extended the IIOP (Internet InterORB Protocol) implementation of COOL, such that application objects are able to communicate and negotiate their QoS requirements with their peer objects and with Da CaPo. The extended IIOP is called QIOP (QoS InterORB Protocol). The task of Da CaPo entities is to select and negotiate for each binding a protocol configuration that supports the particular QoS requirements and to manage the exchange of application data through this protocol. The latter task includes typical connection management tasks like connection establishment, monitoring, and connection release.

Since QIOP represents only a simple and fixed solution for QoS negotiation at the object layer, we are currently
working on a new version of the MULTE-ORB in which Da CaPo implements also the object-layer. This version will be able to dynamically choose components that implement for example IIOP, QIOP, and special multimedia stream adapters. However, the main shortcoming in signalling in the MULTE-ORB is closely tied to the design and implementation of signalling in Da CaPo. Therefore, we briefly introduce some implementation issues of Da CaPo with main focus on signalling in the next section. Furthermore, we discuss the shortcomings of this solution we have experienced when implementing and using RSVP (Resource Reservation Protocol) for resource reservation in Da CaPo.

A. Signalling in Da CaPo

The architecture and implementation of Da CaPo distinguishes between two types of communication protocols, the data protocol and the connection management protocol. The data protocol is responsible to handle application data and transfer it to the peer application. It is implemented by a set of modules, i.e., implementations of protocol functions, that are selected at runtime and dynamically bound together. Here two modules have a special role, in each configuration of a data protocol: the T-module implements a Da CaPo conform interface to arbitrary network services, like ATM, IP, ISDN, etc. and the A-module encapsulates a simple application, e.g., video source and sink, or implements a service access point for a higher layer entity, e.g., application or QIOP in MULTE-ORB 1.0.

Figure 1 illustrates the architecture with the application using the data protocol and connection management protocol to communicate. The application as a generic name for the user of the protocol services, it may be the application itself, or a middleware entity (i.e., stub or IIOP module). The data protocol entity contains a protocol graph with the A-module towards the application and the T-module towards the network. In between the A- and T-modules other modules are configured, that implement the necessary protocol functions to fulfill the application requirements.

A basic design issue of Da CaPo is to perform signalling tasks that are related to the configuration and management of the data protocol out-band. These signalling tasks are implemented in the connection management protocol. The two main tasks of the connection management protocol are:

- **Protocol negotiation**: communicating Da CaPo peers have to use a common protocol configuration. Applications can choose between three negotiation types: unilateral, bilateral, and combined. In the unilateral negotiation, the initiating peer determines a protocol configuration based on local information and informs the peer(s) about the result. The peers can either accept the configuration or abort. In contrast to this “take it or leave it approach”, the bilateral negotiation aims at finding a globally best configuration by exchanging and comparing lists of configurations. The bilateral negotiation introduces considerable delay, but finds a configuration that is best for all peers. The combined negotiation combines the benefits of unilateral and bilateral negotiation by (1) starting with a unilateral negotiation, (2) establishing a data connection based on the result of the unilateral negotiation, (3) performing an bilateral negotiation while the application peers exchange data, and (4) exchanging the data protocol with the result of the bilateral negotiation, i.e., performing a re-configuration.

- **Connection management**: the connection management protocol coordinates establishment, re-configuration and release of data connections, by (1) exchanging corresponding protocol data units between the connection management peers and (2) using local services at each peer system to dynamically link and initialize, i.e., start, a data protocol respectively to stop and remove it.

Since all these tasks and options are implemented in a single protocol, i.e., the connection management protocol, it’s implementation ended up to be quite complex. The reasons for why the data protocol is configurable and the connection management protocol is static and implemented in a single protocol are found in the motivation for the first version of Da CaPo. The focus of the first version of Da CaPo was functionality and how to build light-weight protocols. Also, how far it is possible to decouple the data protocol from the connection management and signalling tasks was not identified and evaluated. In current development of Da CaPo also non-functional requirements, that is QoS and resources, are also of interest, motivating the management toolkit described in this paper.

B. Case: Using RSVP in MULTE-ORB

In order to support QoS for bindings over the Internet, we have integrated the Resource Reservation Protocol (RSVP) in the MULTE-ORB 1.0. We have used the
The parallel execution of protocol negotiation and resource reservation would improve the performance.

The connection management protocol currently requires that both signalling and resource reservation be done before resource reservation can be initiated. This results from the fact that these signalling tasks are not always the best solution, because RSVP signalling is independent of the actual data exchange between applications. However, the connection management protocol is already quite complex. Extending it with one particular protocol for resource reservation is not appropriate. The MULTE-ORB must also support different solutions for resource reservation, like Differentiated Services in the Internet or ATM. Obviously, it is neither meaningful nor feasible to extend the connection management protocol for every new type of resource reservation protocol that should be supported.

Therefore, RSVP has to be integrated in the data protocol. In order to keep signalling for resource reservation at a consistent location in the data protocol, we decided to integrate RSVP in T-modules. This is the same solution as implemented in the ATM T-modules. Due to the available ATM APIs there is no other choice for ATM T-modules than to provide this integrated solution for resource reservation and virtual path and virtual circuit establishment.

We have combined RSVP and UDP in a T-module. This means that the resource reservation is integrated in the data path of the protocol. The T-module is responsible for both signalling and data processing tasks. The reservation is initiated when the T-module is initialized. Refresh-messages and admission control are transparently performed by the RSVP-daemon. So far, we have tested this solution on a peer-to-peer basis in a local subnet. Our experiments have identified several shortcomings with this solution, which are mainly based on the fact that the integration of signalling in the data protocol creates unwanted dependencies. Consider a scenario in which an application does not require any guarantees for QoS. Consequently, the result of the protocol negotiation is a configuration without RSVP. If the application changes its requirements later on and request resource reservation, the data protocol has to be reconfigured to include RSVP, even if RSVP does not handle any of the packets that are exchanged by the data protocol peers. Another disadvantage of this solution is that the data graph has to be agreed upon and built before resource reservation can be initiated. This results in increased establishment delay and reduced performance. The parallel execution of protocol negotiation and resource reservation would improve the performance.

Additionally, there are several examples where the particular configuration of the data protocol should be based on the reservation results. For example, if the throughput requirement cannot be met, the sending side would typically compress data prior to transmission.

In summary, integrating resource reservation into the data protocol determines that protocol negotiation has to be done before resource reservation, but this order of executing these signalling tasks is not always the best. The integration of other signalling tasks, like QoS negotiation, etc., would create the same type of unwanted dependencies between data protocol and connection manager. In order to avoid these dependencies, signalling must be decoupled from the data protocol. Therefore, we have replaced the connection management protocol with a management toolkit that is able to:

- support multiple signalling protocols for resource reservation and protocols for other tasks like QoS negotiation, QoS mapping, admission control, etc.,
- easily integrate new signalling and management protocols, and
- execute the protocols in arbitrary (user defined) order.

IV. Toolkit Architecture

In this section, we use a two step approach to explain the design of the management toolkit. First, we give an overview over the general architecture and the interaction between the toolkit and the other MULTE-ORB components. Second, we present a detailed descriptions of the internals of the toolkit.

A. High-level Architecture

The overall architecture of the MULTE-ORB is illustrated in Figure 2. The entities are depicted as boxes, while interfaces and interactions between entities are represented by arrows between boxes. The application entity is either some kind of application or a higher-level middleware component, like IIOP, QIOP or the stub and object adapter. The management toolkit and the data protocol is part of the ORB core of the middleware, using CORBA terminology. The operating system (OS) exposes different services, like access to resources and OS functionality, through well defined interfaces (APIs). One class of these APIs provides access to network services, but we regard the network as a separate entity in the architecture, because it's resources and services are distributed in the network itself. The data protocol and management toolkit use the network through the OS APIs to communicate with their peers. In the rest of this subsection, we describe the interactions between:

- application and lower layer, i.e., management toolkit and data protocol,
- management toolkit and data protocol,
- management toolkit and OS, and
- network and the higher layer, i.e., management toolkit and data protocol.

The application can access the system via two different interfaces: The DA interface provided by the data protocol and the MA interface implemented by the management toolkit.
The data protocol consists of the Da CaPo runtime system and protocol graph(s), as briefly described in Section III. The A-module implements the DA interface. The DA interface typically offers engineering support for computational interfaces with operational, signal and stream semantics as defined in Reference Model of Open Distributed Processing (RM-ODP) [7].

Methods in the DA interface are regular calls from the application to the data protocol. To appropriately serve multimedia applications, DA also supports flows of continuous data. In addition, applications can implement upcall methods in order to receive events or notification from the underlying entity. Examples of upcalls over the DA interface include the provision of acknowledgements to application level components, control information regarding multimedia flows from streaming devices, etc.

The MA interface is used for exchanging control and management information between the application and the management toolkit, as shown in Figure 2. The methods in MA include methods to specify QoS requirements, initiate connections, release connections, subscribe to monitoring events and so on.

Upcalls to the application over the MA interface include receiving events from higher-order monitors, adaption demands, error reports, etc.

Regarding the format of the QoS specifications, the MULTE QoS model [8] supports specifications in essentially three different formats or levels, as illustrated in Figure 3. The most abstract and least flexible approach is to use an Application Specific Adapter (ASA), which restricts the QoS parameters to a given set depending on the application, and the binding policies are provided as fixed BF, BM and BD scripts. A more flexible way is to use the language B to specify the QoS requirements and binding policies, the language B is part of our binding framework. The QoS parameters in B are typically application level QoS and media format descriptions. In addition, policies and constraints on the binding can be specified [8]. Finally, the application developer may choose to specify requirements using the QoS mediator language M, which essentially is an low-level abstraction over network and OS resources. The application developer can gain fine grained control by using M, but also needs large amount of knowledge about the system and resources.

The QoS specification may be statically annotated to the interface specification of a given DA interface or given as parameters in a method call over the MA interface. In addition, dynamic QoS requirements may be given by the application in order to adapt or simply adjust the requirements during the lifetime of the connection using the MA interface. In this way, the toolkit supports both declarative and procedural specification and adaption of the QoS requirements.

The basic design decision for the management toolkit is to perform out-band signalling. However, when implementing certain protocol functions in Da CaPo, the specifications may require signalling to take place between the peer modules. This is not prohibited and the modules in
the data protocol can perform peer-to-peer signalling, a kind of in-band signalling. In-band signalling is useful in some domains, for instance if signalling has strict latency requirements as when dealing with handovers in mobile communication.

The interface MD includes methods for the management toolkit implemented by the data protocol and vice versa. The interactions between the management toolkit and the data protocol can be classified according to the following tasks:

1. (re-)configuration data protocol,
2. parametrization of modules,
3. resource reservation,
4. connection management,
5. synchronization of data- and signalling protocols, and
6. monitoring events.

The first five tasks in the above list correspond to interactions that are always initiated by the management toolkit. Configuration and re-configuration of the data protocol means to arrange the modules in the protocol graph, deciding the dependencies between the modules. Parametrization of modules takes care of initializing the modules with setting appropriate parameters, examples of parameters are internal buffer sizes, timeout values, address information, and so on. The resource reservation task in the management toolkit operates in the scope of the whole protocol graph, that includes size and properties of shared buffer spaces, thread scheduling policy, and other common resource configuration. In addition, resources are reserved in (some of) the modules. Such internal resources includes threads and buffers. Coordination of the connection management means to start and stop the transmission of data over the data protocol.

In order to carry out dynamic QoS management, the delivered QoS and resource utilization is monitored. In case the monitoring components reports back that the monitored parameters are below some given threshold, the management toolkit may adapt the data protocol. This is performed either by changing parameter values in the modules of the data protocol or, if more radical change is needed, re-configure the data protocol.

Interaction between the management toolkit and the operating system takes place over the MO interface. The management toolkit handles the interaction with the OS on behalf of the application and the data protocol for QoS-controlled services. There is no way for the management toolkit to prohibit applications to directly access the native OS system calls and services, but in order to benefit from the QoS-controlled services the interfaces to the management toolkit have to be used. An example of this is allocation of memory, where the application is free to allocate buffer space using systems calls as malloc(). But in order to allocate buffers in pinned memory that will not be swapped or paged to disk, the application will request this kind of memory from the management toolkit.

The MO interface consists of a set of wrappers over native OS resources. These wrappers are abstractions over resources like memory, processing time (threads), etc. Various resource managers are candidates for inclusion in the management toolkit, examples are memory managers for buffer abstractions with different properties and user level schedulers for threads.

The network entity in Figure 2 represents an end-to-end communication service, such as IP, an ATM adaption layer, GSM data service and so on. The network is accessed by the management toolkit via the MN interface and by the data protocol via the DN interface.

The interface DN is implemented by the T-module in the Da CaPo module graph part of the data protocol. The T-module represents a wrapper or “glue” between different network technologies and APIs, and the module graph. The management toolkit has a similar module implementing the MN interface. These network near modules masks out differences in the underlying network, providing a uniform API “upwards” to the module graph and signalling modules respectively. The modules offers a send and receive interface to the module graph and signalling modules in a Da CaPo consistent way. In the data protocol it is possible to use the same T-module for several module graphs, then (de-)multiplexing is performed in the T-module. This is the only (de-)multiplexing performed in the module graph. The T-module in the management toolkit has to multiplex and de-multiplex messages in order to deliver the information to the right management module. In order to achieve this, the management modules tag the messages they send with an unique ID.
The data protocol and management toolkit is independent regarding what network service (protocol) to use. So, in a given case the data protocol may use UDP/IP as a network service, and the communication between peers of tasks in the management toolkit may use Simple Control Transmission Protocol (SCTP) for transport.

B. Low-level Architecture

In the previous subsection, the overall picture was presented from a high-level perspective. In this section, we look into the internals of the management toolkit. As we see in Figure 2, the management toolkit consists of different management task classes, a coordinator and a number of scripts to process.

The following list comprises those signalling or management tasks we currently consider in the management toolkit, the tasks can be grouped into the following classes:
- QoS parameter mapping
- resource pre-reservation
- negotiation of QoS
- resource reservation
- negotiation of component configuration
- connection management
- monitoring

As discussed earlier, tasks for QoS enforcement and resource management are additional candidates for adoption into the management toolkit (cf. Section IV-A). Now we discuss in general, and detail the description of the tasks listed above.

It is generally possible to find different mechanisms (algorithms or strategies) for each of these management tasks. This is equivalent to representing each task as a class and the different mechanisms as sub-classes (specializations). The actual implementations of these sub-classes are used as instances at runtime, and referred to as management modules.

The task QoS parameter mapping performs mapping from a number of application specific formats or languages to the QoS mediator language M and the binding framework language B in MULTE. The sub-classes of resource pre-reservation will perform optimistic reservation or statistical prediction for statistical QoS guarantees, reserve for the worst case in case of guaranteed QoS, and so on. In order to reach consensus among the communication parties, there is a need to perform negotiation. In the current version we have identified the need to negotiate QoS requirements, component configuration (protocol graph), and policies. The negotiation of QoS task may use strategies as triangular, bilateral or unilateral negotiation, each possibly with different semantics according to the selected implementation of the QoS negotiation task and the (application and system) policies. Semantics from ISO QoS-F [8] and OSI-95 [9] include for instance negotiation with bounded target, and parameters representing maximal, threshold and compulsory QoS. Negotiation of Da CaPo configuration is needed in order to determine if all communicating systems are able to instantiate a common protocol graph. Some systems may not have the modules available (i.e., not implemented) or may need to use modules affecting the resources less. The QoS requirements influence what kind of connection management strategy or algorithm we may use, specially QoS parameters as connection setup and release delay and required reliability guides the selection of the implementation of the connection management task in the management toolkit. Examples of connection management strategies include three-way handshake, two-way handshake, fast connection setup and implicit connection setup. In order to do dynamic QoS management, monitoring of QoS and resources is needed. Monitoring modules in the Da CaPo module graph, notifications from the network and OS provide low-level monitoring events. These events have to be processed according to policies and rules, and these higher-level events are used by the QoS controllers in a feedback control loop and possibly by the application for adaption purposes.

The signalling modules implement the actual mechanisms to be used in the policies. Policies are implemented by scripts given to the coordinator component of the management toolkit. The coordinator are given three scripts, the scripts implements the BF, BM and BD components of the binding protocol (Section IV-A).

As illustrated in Figure 4 the script language offers imperatives like loops, tests and so on. From an architectural point of view, it is of no importance which language is used for the scripts if the necessary language constructs are supported, and the coordinator is able to execute the script. For example, in our first prototype binary scripts (compiled C++ code) and interactive “shell like” execution, and the language Python will be supported in the next versions.
The coordinator maintains the state of the connection, and takes care of transition from phase to phase (i.e., script to script), this sequence is shown in Figure 5.

The scripts call methods in the signalling modules, Figure 6 illustrates the concept. In order to do this, the different signalling module interfaces have to implement well known interfaces, these interfaces typically are derived or specialized version according to the class hierarchy of the signalling modules.

The signalling strategies (scripts) define which classes and in which order to invoke or activate the management modules. The synchronization of the signalling with the data protocol have to be controlled by the concrete modules selected during the instantiation of the BF, BM and BD scripts. Examples of synchronization interactions between management toolkit and data protocol are these two cases: An application with very strict latency requirements and more relaxed requirements on throughput guarantees can be supported best by establishing first a connection with the data protocol and performing afterwards QoS negotiation and resource reservation. In contrast, most realtime applications require to perform QoS negotiation, admission control and resource reservation before the data protocol is set-up. In order to tackle this, the coordinator maintains the state of the data protocol, in our case data protocol “running” and “stopped”.

The applications provide QoS requirements using the method setQoSParameter() offered in the MA interface. The same solution is offered the application via the interface stubs in the QIOP prototype implementation. When setQoSParameter() is called the QoS parameter structure is directed to the right QoS mapper, if necessary, and stored in QoS objects. The QoS information is stored in these QoS objects (one for M and B level parameters), and shared among signalling modules. The information about the protocol graph (state, composition and so on) are stored in a separate object. These QoS and graph objects are accessible via the coordinator.

V. IMPLEMENTATION AND PRELIMINARY EVALUATION

A subset of the management toolkit is implemented as a “proof-of-concept” implementation. The implementation, our first experiences with the prototype, and description of how we will proceed with the evaluation and verification of the architecture are described in this section.

A. Management Toolkit 1.0

The management toolkit 1.0 is implemented in C++ on a Solaris 2.6 platform. To limit the programming overhead and simplify the testing, some of the modules were implemented in a simplified version only. A simplified version implements a subset of the functionality of the algorithm it offers, or simply contains the code necessary to do logging. However, the module interfaces and the interaction between the entities were made authentic.

As the existing Da CaPo version is implemented in C, no object orientation is used. Consequently there is no well defined interface suitable for signalling extensions. A full integration of the toolkit in Da CaPo would require ad hoc programming to conform to the existing implementation. It therefore seemed undesirable to integrate the toolkit into the existing Da CaPo implementation at this stage. Instead we provided an object oriented wrapper around Da CaPo when developing the management toolkit. A simplified object structure was created to simulate the behavior of the data modules, data graph and protocol entities in the new Da CaPo version. Additionally, the original interfaces of the data modules and T-modules were extended to allow some interaction with the toolkit.

The signalling modules are controlled by a coordinating module. The coordinator supervises the signalling activity according to a specified policy, i.e., script. Version 1.0 uses two different types of scripts. One is hardcoded in C++ and linked at compile time into the coordinator. The other is generated by the user interactively at runtime in a shell-like fashion. The main advantage of the latter, is that new configurations can be tested without recompiling the toolkit. Since these two extreme choices for implementation work quite well, we are convinced that a more moderate approach with interpreted languages like Python and Perl, will work without problems.

The QoS parameters of the different QoS specifications are encapsulated in separate object. Every object is a subclass of a common QoS superclass. This enables easy integration of new specifications, because a new QoS object is treated simply as an instance of the superclass. The peers can therefore negotiate every possible QoS specification through the same interface. The implementation supports three different subclasses; one for QoS specified using the B language, one for M-QoS (QoS in the M language) and one for description of the data graph configuration. The QoS objects are maintained by the coordinator. In a parallel system, this approach requires concurrency control when several modules wish to access the QoS parameters simultaneously.
The coordinator is responsible for creating, deleting and activating the signalling modules. Each of the modules is instances of the same superclass Module, this is according to our architecture described in Section IV-B. The signalling modules are implemented as specializations of Module using inheritance, examples of implemented modules includes a QoS mapper Mapper, a resource reservation module Reserve, and a negotiation module Negotiate. Implemented specializations of Reserve includes a module deploying RSVP.

The management toolkit uses a network entity to handle network traffic and operating system signals. The module is responsible for addressing and transmission of signalling packets, and to secure a reliable signalling channel between the communicating peers. How to handle operating system signals efficiently in both data and management planes without introducing an additional level of demultiplexing, is an issue that we currently investigate.

In order to limit consistency problems and ensure independence between the modules, no module has knowledge of the others. If the negotiation module wants to transmit a packet, it has to ask the coordinator for a valid reference to the network entity. The reference is used to deliver the packet. This solution requires minimal overhead in case of reconfiguration.

B. Experiences with this implementation

The 1.0 version was tested using the dynamic policy approach. This allowed the user to combine the signalling modules dynamically, and observe the effects on the system. Every function call was logged, along with the packet contents, interrupts and QoS-values.

A given sequence of signalling tasks is called a permutation. Each permutation has certain properties, like ‘low establishment delay’ or ‘guaranteed bandwidth’. If connection establishment is done before resource reservation, the establishment delay will be relatively low. If negotiation and resource reservation are performed prior to connection establishment, the delay will increase.

The intension was to verify that the toolkit could meet different application needs. It was therefore crucial to see if relevant permutations could be executed, whether the policy was realized or not, and how change of QoS-parameters affected the system.

Permutations tested include:

- QoS mapping - QoS negotiation (M level)
- QoS negotiation (B level) - QoS mapping - QoS negotiation (M level)
- Resource pre-reservation - QoS negotiation (M level)
- QoS negotiation (B level) - QoS mapping - Resource reservation

The test results show that the permutations behave as expected. Faults occur when illegal permutations are executed, i.e. the network module is not created prior to negotiation, or when QoS is not mapped to an appropriate syntax (M) before use.

Typically, performance is an important part of the evaluation of systems. In our case, it is not of primary concern, because the experiences with Da CaPo have shown that encapsulation of single protocols in modules and their flexible configuration does not have a severe negative impact on performance. However, encapsulation enables flexibility which is otherwise not possible.

VI. Conclusions

The basic goals of the MULTTE-ORB are to provide dynamic QoS support, enable policy controlled management of bindings, and allow easy integration of new components into the ORB. In order to reach these goals, we have combined in the MULTTE-ORB 1.0 the protocol configuration framework Da CaPo with the COOL ORB and QIOP. The experiences with MULTTE-ORB 1.0 have shown that flexibility of the data path is not sufficient. Additionality, flexibility is needed for the management of binding, i.e., flexible signalling and management protocols. Since Da CaPo is not able to support this type of flexibility, we have developed a management toolkit that supports an extensible set of signalling protocols, allows to easily integrate new signalling protocols, and can perform the different signalling protocols in any given order. Main elements in the architecture are scripts that define which management protocols have to be used in which order during the three lifetimes of a binding. By implementing the central elements of the management toolkit we have demonstrated that our claims for flexible signalling are fulfilled by the management toolkit.

To the best of our knowledge, there is no other approach or system that supports this type of flexibility for managing bindings. Other flexible middleware solutions, like Da CaPo++ and dynamicTAO, have the same basic problems as MULTTE-ORB 1.0, signalling and management are solved in a fixed monolithical approach. Other approaches that are more concerned with open signalling include the research projects associated to the international OPENSIG working group. Prominent approaches includes XBind, NCAM at Cambridge University and the work with Open Network Control at Lancaster University. Standardization efforts like TINA and ReTINA, associated to TINA-C, and the IEEE P.1520 working group are also contributing to the field. Associated to the area of open signalling and architectures are research in active networks, and specification of open interfaces for network programming such as Parlay.

Our ongoing work on configurable signalling includes working on a prototype that integrates a Python interpreter in the coordinator entity of the management toolkit to allow maximum flexibility for application and users to define policies at run-time. Furthermore, we implement additional signalling modules to perform more experiments that are based on concrete applications scenarios and real-life problems.

The binding framework in MULTTE allows to recursively open up bindings down to a primitive binding, which typ-

*http://www.comet.columbia.edu/opensig/*
ically is an IP or ATM binding. With active networks it is also possible to open up for instance IP bindings. However, in this work we have focussed on signalling that is performed between end-systems using standard mechanisms and protocols in the network as RSVP in IP networks and UNI/NNI signalling in ATM networks. Future work in the MULTe project includes to extend the flexibility of the data and management protocols in end-systems with dynamically downloadable and flexible protocol elements on active networks nodes. The first case to investigate is how to solve heterogeneity issues between participants in a binding, this includes heterogeneity in data format, capacity and QoS in general. The idea is to insert filters in the network, that is to dynamically place filter components in a multicast tree in an optimal manner. One example of the elements to be inserted in the network nodes is transcoders for audio and video data (e.g., MPEG to H.263), other examples are capacity or content filters for connecting PDAs (Personal Digital Assistants) and cellular phones to the binding. For these experimental studies we will use small embedded PC’s, so called Smart Port Cards (SPC) \[20\], in gigabit ATM switches developed at Washington University St. Louis \[21\].

**REFERENCES**


